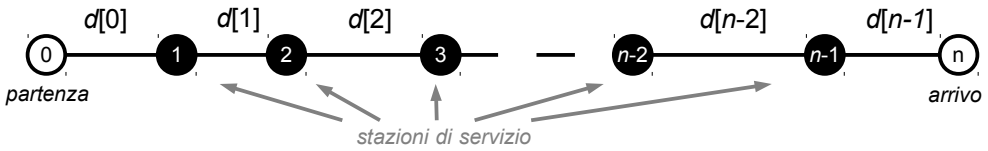
Corso di Algoritmi e Strutture di Dati

Esercizi

Esercizio 1. Un'auto può percorrere K Km con un litro di carburante, e il serbatoio ha una capacità di C litri. Tale auto deve percorrere un tragitto lungo il quale si trovano $n + 1$ aree di sosta indicate con 0, 1, … n, con $n \geq 1$. L'area di sosta 0 si trova all'inizio della strada, mentre l'area di sosta n si trova alla fine. Indichiamo con $d[i]$ la distanza in Km tra le aree di sosta i e $i + 1$. Nelle n - 2 aree di sosta intermedie {1, 2, … n - 1} si trovano delle stazioni di servizio nelle quali è possibile fare il pieno (vedi figura).



Tutte le distanze e i valori di K e C sono numeri reali positivi. La auto parte dall'area 0 con il serbatoio pieno, e si sposta lungo la strada in direzione dell'area n senza mai tornare indietro.
Progettare un algoritmo in grado di calcolare il numero minimo di fermate che sono necessarie per fare il pieno e raggiungere l'area di servizio n senza restare a secco per strada, se ciò è possibile. Nel caso in cui la destinazione non sia in alcun modo raggiungibile senza restare senza carburante, l'algoritmo restituisce -1.

Soluzione.

```
MinFermate( real d[0..n - 1], real K, real C ) → integer
    real res ← K * C;
    integer i ← 0, f ← 0;
    while ( i < n ) do
        // res è il carburante residuo che mi rimane una volta arrivato alla stazione i.
        if ( res < d[i] ) then
            res ← C * K;
            f ← f + 1;
        endif
        res ← res – d[i];
        if ( res < 0 ) then
            return -1;
        endif
        i ← i + 1;
    endwhile
    return f;
```

Tutte le operazione hanno costo costante. Il corpo del ciclo while viene eseguito n volte. Il costo computazionale risulta quindi essere $\mathbf{T(n) = \Theta(n)}$.

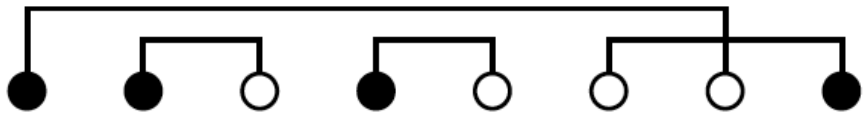
Esercizio 2. Disponiamo di un tubo metallico di lunghezza L. Da questo tubo vogliamo ottenere al più n segmenti più corti, aventi rispettivamente lunghezza $S[1]$, $S[2]$, …, $S[n]$. Il tubo viene segato sempre a partire da una delle estremità, quindi ogni taglio riduce la sua lunghezza della misura asportata. Scrivere un algoritmo efficiente per determinare il numero massimo di segmenti che è possibile ottenere. Formalmente, tra tutti i sottoinsiemi degli n segmenti la cui lunghezza complessiva sia minore o uguale a L, vogliamo determinarne uno con cardinalità massima. Determinare il costo computazionale dell'algoritmo proposto.

Soluzione. Ordiniamo le sezioni in senso *non decrescente* rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima. Procediamo quindi a segare prima il segmento più corto, poi quello successivo e così via finché possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento). Lo pseudocodice può essere scritto in questo modo:

```
MaxNumSezioni( integer L, integer S[1..n] ) → integer
   OrdinaCrescente(S);
   integer i ← 1;
   while ( i ≤ n and L ≥ S[i] ) do
        L ← L - S[i];      // diminuisce la lunghezza residua
        i ← i + 1;
   endwhile
   return i - 1;
```

L'operazione di ordinamento può essere fatta in tempo $\Theta(n \log n)$ usando un algoritmo di ordinamento generico. Il successivo ciclo while ha costo $\Theta(n)$ nel caso peggiore. Il costo complessivo dell'algoritmo (assumendo di ordinare in tempo pseudolineare) risulta quindi $\Theta(n \log n)$.
Si noti che l'algoritmo di cui sopra restituisce l'output corretto sia nel caso in cui gli n segmenti abbiano complessivamente lunghezza minore o uguale a L, sia nel caso opposto in cui nessuno abbia lunghezza minore o uguale a L (in questo caso l'algoritmo restituisce zero).

Esercizio 3. Lungo una linea, a distanze costanti (che per comodità indichiamo con distanza 1), sono presenti n punti neri ed n punti bianchi. È necessario collegare ogni punto nero ad un corrispondente punto bianco tramite fili; ad ogni punto deve essere collegato uno ed un solo filo. Scrivere un algoritmo efficiente per determinare la quantità minima di filo necessaria. Determinare il costo computazionale dell'algoritmo proposto.
A titolo di esempio, nell'immagine sotto viene riportata una istanza del problema con 4 punti neri e 4 punti bianchi, ed un corrispondente collegamento di punti che richiede l'uso di una lunghezza complessiva di filo pari a 10. Si noti che tale collegamento non è ottimale, in quanto sarebbe possibile usare una lunghezza complessiva pari a 8.



Soluzione. È possibile risolvere il problema con un semplice algoritmo greedy, leggendo i punti da sinistra a destra, e collegando ogni punto incontrato al primo fra i successivi di colore diverso. Nel caso in cui ci siano più punti di medesimo colore che si possono collegare ad un successivo punto di colore diverso, si dà priorità a quello più lontano.

```
algoritmo CollegaPunti( bool p[1..2n] )          // true = bianco,  false = nero
   int i, filo ← 0;
   queue bianchi ← new queue(),
   queue neri ← new queue();
   for i ← 1 to 2n do
        if (p[i]) then                           // i-esimo punto bianco
             if (neri.empty())  then
                  bianchi.enqueue(i);
             else
                  filo ← filo + (i – neri.dequeue());   // collega ad un precedente nero
             endif
        else                                     // i-esimo punto nero
             if (bianchi.empty()) then
                  neri.enqueue(i);
             else
                  filo ← filo + (i – bianchi.dequeue());   // collega ad un precedente bianco
             endif
        endif
   endfor
   print "Lunghezza minima filo: " + filo;
```

Considerando costo costante per le operazioni di *new*, *empty*, *enqueue* e *dequeue* sulle code, il costo dell'algoritmo è $\Theta(n)$, visto che il corpo del ciclo while viene eseguito 2n volte,

Esercizio 4 Supponiamo di avere $n \geq 1$ oggetti, ciascuno etichettato con un numero da 1 a n; l'oggetto i-esimo ha peso $p[i] > 0$. Questi oggetti vanno inseriti all'interno di scatoloni identici, disponibili in numero illimitato, ciascuno in grado di contenere un numero arbitrario di oggetti purché il loro peso complessivo sia minore o uguale a C. Si può assumere che tutti gli oggetti abbiano peso minore o uguale a C. I pesi sono valori reali arbitrari. Vogliamo definire un algoritmo che disponga gli oggetti negli scatoloni in modo da cercare di minimizzare il numero di scatoloni utilizzati. Questo genere di problema è noto col nome di *bin packing problem* ed è computazionalmente molto complesso nel caso generale; di conseguenza, ci accontentiamo di un algoritmo semplice che produca una soluzione non necessariamente ottima.

1. Scrivere un algoritmo basato sul paradigma greedy che, dato il vettore dei pesi $p[1..n]$ e il valore C, restituisce il numero di scatoloni che vengono utilizzati.
2. Calcolare il costo computazionale dell'algoritmo proposto.

Soluzione. Un algoritmo greedy molto semplice consiste nel considerare tutti gli oggetti, nell'ordine in cui sono dati. Per ogni oggetto, si controlla se può essere inserito nello scatolone corrente senza superare il limite di peso. Se ciò non è possibile, si prende un nuovo scatolone e lo si inizia a riempire.

```
algoritmo SCATOLONI( array p[1..n] di double, double C ) → int
    int ns ← 0;                    // numero di scatoloni utilizzati
    int i ← 1;
    while ( i ≤ n ) do
        ns ← ns + 1;               // iniziamo a riempire un nuovo scatolone
        double Cres ← C;           // capacità residua dello scatolone corrente
        while ( i ≤ n && Cres ≥ p[i] ) do
            Cres ← Cres – p[i];
            i ← i + 1;
        endwhile
    endwhile
    return ns;
```

L'algoritmo fa uso della variabile intera ns, che mantiene il numero di scatoloni utilizzati, e della variabile reale Cres che indica la capacità residua dello scatolone corrente. Se la capacità residua Cres supera il peso $p[i]$ dell'oggetto i-esimo, allora tale oggetto può essere inserito nello scatolone; si provvede quindi a decrementare Cres di $p[i]$ e si passa all'oggetto successivo. Quando Cres diventa inferiore a $p[i]$, allora l'oggetto i non trova posto nello scatolone corrente, e si inizia a riempire il successivo.
Il costo computazionale dell'algoritmo proposto è $\mathbf{O(n)}$.

L'algoritmo qui sopra è molto efficiente, ma piuttosto "stupido": è facile pensare a varianti che in molti casi (ma non sempre) ottengono risultati migliori, a scapito della complessità. Ne discutiamo brevemente due:

1. ad ogni passo si inserisce l'oggetto nella prima scatola in grado di contenerlo;
2. si ordinano gli oggetti per pesi decrescenti, in ogni scatola si inseriscono oggetti pesanti (prendendoli da inizio sequenza) finché possibile, poi oggetti leggeri (prendendoli da fine sequenza) finché possibile.

Si vede facilmente che la complessità del primo algoritmo è $\mathbf{O(n^2)}$ e quella del secondo è $\mathbf{O(n \log n)}$. Un confronto preciso tra le qualità delle soluzioni proposte dai diversi algoritmi è molto complesso, visto che si deve considerare una distribuzione di probabilità per i valori ed eseguire ragionamenti probabilistici.

Gli algoritmi indicati sopra non garantiscono di trovare il numero minimo di scatole. Trovare tale numero minimo coincide, come detto nel testo dell'esercizio, a risolvere un problema noto in letteratura come *bin packing problem*. Tale problema risulta essere NP-completo, cosa che implica che al momento non si conoscono algoritmi con costo computazionale polinomiale capaci di risolvere tale problema.